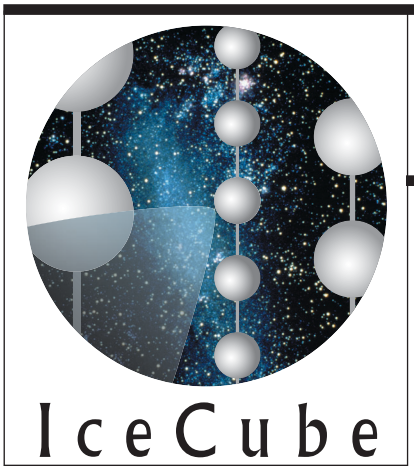

IceCube Software

An Introduction to the IceCube Software Development Environment

Simon Patton L.B.N.L.

This document provides an introduction to the concepts and tools that make up the IceCube Software Development Environment. While concentrating on the “Personal” aspect of this environment it also gives an overview of how this system works in conjunction with the “Collaborative” environment.

Last Modified: April 23, 2003 10:48 am

Version:0.8

1.0 Introduction

1.1 Purpose

The purpose of this document is to introduce the concepts and tools that make up the IceCube Software Development Environment. However, it is not intended to be an authoritative reference on any of these tools. That information can be obtained from their individual documentation. (See 1.3 ‘‘References’’). The intention is that by the end of this document the reader can feel comfortable using the IceCube Software Development Environment and can carry out all the basic tasks needed for simple software development.

The intended audience for this document is anyone who will be using the IceCube Software Development Environment. This includes, but is not restricted to, developers and users of “production” IceCube software. While it is not required that developers of non-production IceCube software use this environment, every effort has been made to make the environment easy to use and they are encouraged to try it out.

1.2 Scope

The scope of this document is to provide enough information about the concepts and tools that make up the IceCube Software Development Environment so that someone who is new to this environment can get up and running and make productive contributions to the “production” IceCube software. To this end this document will also discuss how the tools introduced here help with the implementation of the software process. These tools should be followed for all “production” software and are recommended for “non-production” software.

Production software is defined as that code which is run by people on behalf of the collaboration. This covers all the code used to acquire, process, and transport data taken at the South Pole until is it made available to physicists via access to the Data Warehouse. It also covers all the code needed to generate and process collaboration-approved simulated data sets. A physicist’s individual analysis is not considered to be production code.

This document concentrates on the “personal” software development environment, as that contains those components of the IceCube Software Development Environment that an individual developer will most commonly use. However, this document will also outline the “collaborative” software development environment so that an individual developer can see how the two different environments interact. It also highlights those parts of the collaborative environment that provide feedback to a developer on the state of the code in the context of the complete “production” IceCube Software effort.

This document also concentrates on the development of Java code as this is the main language that will be used to develop much of IceCube’s production code. While the IceCube Software Development Environment does support other languages those are covered in a separate document

As noted earlier this document is not intended to be an authoritative reference on any of these tools.

1.3 References

The following references are used throughout this document.

[IceSoft-xxx] IceCube Code Guidelines.

[IceSoft-yyy] BFD User's Guide.

[IceSoft-zzz] FAYE User's Guide.

1.4 Definitions and Acronyms

baseline: An agreed set of code and resources that function together.

head: The version of a project which contains the "highest" production tag, plus any modification which may have been made on top of that tag.

package: A Java concept which defines a naming and accessibility scope.

project: A collection of Java class files and resources that are all packaged into a single jar file.

release: A "baseline" which has been given to the "clients".

work space: A directory in which, or below which, the process of developing a feature or fixing a bug is executed.

1.5 Overview of this Document

The next section gives an outline of the benefits of a common development environment to a Collaboration. It also outlines how the components that can make up such an environment can be assigned to one or both of two distinct categories, namely the personal and collaborative environments.

Following that section the rest of the document in divided into two Parts. The first Part covers the components of the personal environment. It gives the details needed to use these components, both in a stand alone mode and in a baseline mode. The baseline mode is one used to develop "production" IceCube software.

The second Part covers those components of the collaborative environment which do not appear as part of the personal environment. This Part expands how these interact with the personal components and what a developer should monitor to make sure that their software is behaving correctly in the context of the complete "production" software effort.

2.0 A Common Software Development Environment

The first question you should be asking is "Why should I bother using the ICSDE?" (If you have not asked this perhaps I can interest you in some swampland in Florida?) IceCube is providing a Development Environment for its developers and physicists because of the benefits such an environment can bring. These benefits are the following:

- You, the developer or physicist, can concentrate on the problem at hand rather than spending too much time trying to put your own environment together which you then need to integrate with the rest of IceCube's software.

  Moreover, as the person or people responsible for the environment respond to requests from any user for new features, these features can be immediately available to you without your having to spend time working on including them in your own environment.
- You can immediately access and use all the parts of the growing IceCube code.

  If all parts of the IceCube code base use the same environment then it is easy for you to look at, understand and use any part of the system without spending time trying to work out exactly how a particular piece of the system was put together and hold should it be build an used.
- You can get stable foundations upon which you can work on bug fixes and new feature development.

  During the development phases of software products, the code and resources upon which your work depends can be in a constant state of flux. This means that it can be a real challenge to firstly have a stable base upon while you can develop new feature, and secondly, in the case of tracking down bugs, it is not always clear which combination of code exhibits that problem. A development environment can provide tools to avoid these headaches.
- It is easier for developers to transition to other parts of the software effort.

  When a software effort has a common environment the tools and policies a developer learns when working in one area of the effort are immediately applicable when they have to move to a different area of the effort

.

3.0 Responsibilities of a Software Development Environment

This section outlines the responsibilities of a software development environment. As you may have been able to deduce from the benefits that such an environment brings it covers three distinct areas; configuring code, building code, and testing code.

3.1 Configuring Code

When a software effort involves more than one file or more than one developer there is a need to know which files work together and what changes have been made to those files. To this end the software development environment needs to include some sort of "Software Configuration Management" tool. This will allow an individual developer to use and, if permitted, define sets of files that function together. A set of such files is often referred to as a "baseline". A developer normally develops new features and corrects bugs against a specific baseline. The idea of a "baseline" set of files is also used to define the files that make up a final software product. In these cases the features and performance of the entire software product are tested before the baseline is finally defined and in this case the baseline is normally called a "release".

3.2 Building Code

Building on the concept of baselines, it would be really useful for developers to be able to use pre-built files, such as libraries, that have been built for the set of files that make up a baseline, rather than each developer having to create and maintain such a set of files. Therefore, the common build system can offer such a facility. This means that developers can concentrate on their contribution to the product, rather than spending time worrying about whether the rest of the files they use are correct.

Having a common scheme for building contributions also means that it is easy to automate the building of the complete software product. Automation of this complete build means that it can be easily done at regular intervals, for example every night. Thus problems in the current version of the product such as incompatibilities between individual contributions, can be detected at an early stage of development and corrected.

3.3 Testing Code

The benefits raised in the preceding section about having a single approach to building code also apply to a having a common testing scheme. Moreover, having such a scheme helps with the whole development and maintenance cycle.

On an individual level, testing allows a developer to see whether their contribution is performing as expected. It also lets them check that none of the changes they have introduced during development have broken any of the features that were already working. This helps them avoid submitting code which will need to be revisited at a later date. On a collaborative level, testing is the easiest the way to demonstrate that all the individual contributions are functioning together correctly in the completed product and that there are no unexpected side effects from combining individual contributions.

Having a testing scheme also means that when there are problems with a product, it is easy to isolate which area is not behaving as expected and then that problem can be assigned to the correct developer. This helps to stop the other developers from being distracted from their development by hunting for the problem in their contribution. This also allows them to continue to work by using a baseline which tests indicate does not contain the problem.

3.4 Personal and Collaborative Environments

The use of a common software development environment not only means that developers can easily collaborate, but it also means that when all the individual contributions are brought together into a completed software product, a single system can be used to validate it. As has been noted in the preceding sections, the process of validating such a product has different requirements than the process of creating the individual contributions of that product, therefore a common software development environment can be divided into those components necessary for one or both processes. This document will refer to those components that a developer uses to create an individual contribution to the elements of the “personal” software development environment, while those used to validate the entire product will be referred to as the “collaborative” environment.

Part I of this document will cover the elements of the “personal” software development environment, while Part III will cover those components that are part of the “collaborative” software development environment, that have not been covered in Part I.

Part I: The Standalone Personal Software Development Environment

4.0 The Aim of a “Personal” Software Development Environment

The aim of a “personal” software development environment is to provide an individual developer with all the tools they need to create, test and submit code that is part of a larger collaborative software effort. Of course this does not exclude such an environment from being used to create a non-collaborative software product. In fact one of the subsidiary aims of this “personal” software development environment is to help the developer use “good” development practices and, as such, even a non-collaborative software product can benefit from using this environment.

As noted in Section 3.0, a software development environment should provide tools to help configure, build and test contributions to a software product. To this end the “personal” software development environment should include tools that allow a developer to do the following:

- define a baseline upon which they can base their work;
- have access to pre-built files, i.e. libraries, in that baseline (to save space and time during development);
- build their contributions against those pre-built files, rebuilding any files which are dependent on their changes if required;
- run their own tests in the same environment as the collaborative tests;
- provide access to a code archive where they can submit their changes for archiving;
- submit their changes for inclusion in the “next” baseline;

One area that is specifically excluded from these requirements is the prescription of a code editor. The reason for this is simple. Providing that the code produced by the developer conforms to the coding guidelines laid out in IceSoft-xxx and all the other artifacts needed by the code, e.g. `build.xml` files, correctly function in the collaborative environment, then it does not matter how a developer edits their code. Therefore the choice of editor (and possibly IDE) can be left up to the developer.

While this software development environment is designed to be used in a collaborative environment and exploit the idea of baselines, it is easier to start the introduction of the environment using standalone examples, i.e. ones that do not use baselines, to give developers a feel for the environment and development process. Once the basics have been established we can move onto how these techniques work within the context of baselines.

5.0 Example: Building and Running a Simple Project

In this example we'll introduce the idea of a "work space" and how to create one using the `bfd` tool. We'll also introduce the idea of a "project" and the `ant` tool which is used as the build tool in this environment.

5.1 Setting up the Work Space

All your work in a development environment is done in or below a directory that is referred to as the "work space". While it is not a requirement that your work on each feature or bug take place in its own work space, experience shows that this is a wise approach. That way the set of changes associated with each task can easily be separated, which is especially important if one set turns out to be flawed. Therefore, before we can do anything we ned to create a work space in which to work.

In IceCube you create the work space by hand. You start by creating the directory which will function as your work space and then set this to be your working directory.

```
[wile@amce]$ mkdir work
[wile@amce]$ cd work
```

Baseline management and access to the source code archive are tightly coupled responsibilities, therefore there is a single tool, the "Baselined File Development" system (`bfd`) which is used to handle these responsibilities. You will need access to this tool to be able to do any of the examples in this document. If the IceCube software is already installed on your machine you should already have access to this tool. The easiest way to check is the following command:

```
[wile@amce]$ which bfd
```

This will either give you some information about where it is installed or tell you that there is no such command, in which case you should follow the instruction in Appendix A so that you can install your own copy.

When running in a standalone mode `bfd` needs access to a tools directory which contains all the third party software products that IceCube is using. Again, this directory should already be installed by the standard IceCube installation, but if is not then Appendix B details how such a directory can be set up by hand.

Once you have `bfd` and the tools directory installed you should initialize your standalone work space by executing the following command in your work space directory. The final argument of the command is the location of the `tools` directory:

```
[wile@amce]$ bfd init /home/icecube/tools
```

The work space is now ready for use. The first step you should always take when you use a work space is to set up the environmental variables needed for all the tools to function. `sh` users can do this with the following command:

```
[wile@amce]$ . ./setup.sh
```

while `csh` users should use the following:

```
[gromit@wallaby]% source setup.csh
```

5.2 Accessing a Project

A "project" is a collection of source and resource files that are all packaged into a single library. In the case of Java this will be a jar file. This is the basic unit of development and each project is the responsibility of a single developer or, in some of the large cases, a small team of developers. A project should not be confused with the Java idea of a "package" which defines naming and access scopes.

In this example we are going to build and run the code in a project called `wallace`. This project contains a single Java class which is an implementation of a simple counter. To access the current version of this project you need to check out all of its files from the archive. If you have not already done so you need to set this up with the following commands for `sh` users.

```
[wile@amce]$ CVSROOT=:ext:glacier.lbl.gov:/home/icecube/cvsroot
[wile@amce]$ export CVSROOT
[wile@amce]$ CVS_RSH=ssh
[wile@amce]$ export CVS_RSH
```

For `csh` users the following are necessary:

```
[gromit@wallaby] setenv CVSROOT \
                              :ext:glacier.lbl.gov:/home/icecube/cvsroot
[gromit@wallaby] setenv CVS_RSH ssh
```

These environment variables are not set up by the `setup` scripts and so you may want to think about installing them into your login scripts.

You can now check out all of the files in the `wallace` project by using the following command:

```
[wile@amce]$ bfd checkout wallace
```

You should be able to see that the `wallace` project contains a single file Java file, `Counter1.java`, and two organizational files, `project.properties` and `build.xml`. These latter two files are used by the `ant` tool as we shall see in the next section.

5.3 Building and Running the contents of a Project

Most of the standard operations you'll need to do with a project have already been captured in the standard `build.xml` file that comes with any project. Therefore, at least in the early stages of development, you should not need to change this file.

The first step in using the code in a project is to build its jar file. You can do this in three different ways. The most straightforward way is to change into the top directory of the project and use `ant` in that directory. The following commands will do just that for `wallace`:

```
[wile@amce]$ cd wallace
[wile@amce]$ ant lib
```

You should be able to see that the project's jar file, `wallace.jar`, has been created. The `wallace-test.jar` file which is also created will not be covered in this example but will be dealt with later.

Running the program that was created by this process is as simple as the following command line:

```
[wile@amce]$ java  -cp ../lib/wallace.jar \
                            icecube.tools.examples.Counter1 10 3 4
```

This should produce the following output if all has gone well.

```
The current counter value is 3
The current counter value is 2
The current counter value is 5
The final counter value was 6
```

To understand what this program is doing you should take a look at the source code which is in the following file:

```
src/icecube/tools/examples/Counter1.java
```

The directory structure seen in this file name reflects the fact that this class is in the `icecube.tools.examples` package and this is the structure that works best with Java compiler.

To clean up all the files that are generated when a project is built you simply invoke the `clean` target of the `ant` command and return to the work space directory. The following command accomplishes this.

```
[wile@amce]$ ant clean
[wile@amce]$ cd ..
```

5.4 An Alternative Build Procedure

As well as being able to build a project from its own top directory, it is also possible to build it from the work space directory, i.e. one level higher up. This is also the directory where the executables are expected to be executed. To be able to do this you need to specify which project you wish to build as there could be more than one project in the work space. The following command shows how this is done, along with running the resulting program and cleaning up afterwards.

```
[wile@amce]$ ant -DPROJECT=wallace lib
[wile@amce]$ java -cp lib/wallace.jar \
                           icecube.tools.examples.Counter1 10 3 4
[wile@amce]$ ant -DPROJECT=wallace clean
```

You can see, even from the brief example above, that specifying the project for each `ant` command can soon get tedious, therefore there is also a way to specify a default project for `ant` to use if none is given when invoked at the work space level.

```
[wile@amce]$ ant -DPROJECT=wallace defaultProject
{wile@amce]$ ant lib
[wile@amce]$ java -cp lib/wallace.jar \
                           icecube.tools.examples.Counter1 10 3 4
[wile@amce]$ ant clean
```

5.5 Disposing of a Project

Once you have completed work on a project you will probably want to remove it from the work space so you can use it for other things. Later, when you are archiving your

changes, you may also want to check that you have not forgotten to archive any files. To help with this task the `bfd` tool has the `dispose` option. To dispose of `wallace`, for example, the following command, executed in the work space directory, will work.

```
[wile@amce]$ bfd dispose wallace
```

If you now look at your work space using the `ls` command you should see that the `wallace` directory is no longer there. However, you will see that all of the development environment's infrastructure is still present. You can get rid of this using the `dispose` option of `bfd` but without specifying a project.

```
[wile@amce]$ bfd dispose
```

As the output from `bfd` notes "anything left is your own problem."

5.6 Review

In this example we introduced the idea of a work space and a project. The `bfd` tool, which manages a work space, was introduced and the following `bfd` options were used:

init: initializes a work space for standalone mode.

checkout: copies the contents of a project into the work space.

dispose: cleans up a work space.

The `ant` build tool, that builds projects from source files, was also introduced and the following targets used:

lib: builds a projects library.

clean: removes all the files a project has generated during any stage of the build process.

It was also shown how to use `setup.sh` or `setup.csh` to create the right environment in which to work, as well as how to run a simple Java executable.

6.0 Example: Logging Output and Testing Code

The preceding example showed how easy it was to build a project in the development environment. To keep things simple, the `wallace` project did not rely directly on any of the standard tools that are in use in the IceCube environment. (While `bfd` and `ant` were used in the example, the code itself could have been built and run without them.) In this example we will look at code that uses the two most basic tools provided by the environment; a tool for logging output and another which provides a powerful framework in which to develop unit tests. This example will also show how you can work with more than one project in a work space.

6.1 Setting up the Work Space

This example will make use of the `icebucket` library, which contains a set of standard utilities used throughout IceCube software. Therefore the first item of business in this example is to create a work space and build the `icebucket` library. While you

should be able to work out how to do this, the following commands are provided to give you a jump-start.

```
[wile@amce]$ mkdir work
[wile@amce]$ cd work
[wile@amce]$ bfd init /home/icecube/tools
[wile@amce]$ . ./setup.sh
[wile@amce]$ bfd checkout icebucket
[wile@amce]$ ant -DPROJECT=icebucket lib
```

6.2 Building and Running the "gromit" Project

The `gromit` project is the one that will be used in this example and the class containing the `main` routine is now named `Counter2`. By now you should be able to check it out of the archive and build and run it. In case this is still a challenge the following commands are provided to help you along, but this will be the last time you will get such help. If you do not follow them you should really work through the preceding example again before moving on.

```
[wile@amce]$ bfd checkout gromit
[wile@amce]$ cd gromit
[wile@amce]$ ant lib
[wile@amce]$ java -cp ../lib/gromit.jar \
                              icecube.tools.examples.Counter2 10 3 4
```

If we are really lucky the output from the last line should be the same as we got from the `Counter1` execution in the preceding example, namely the following:

```
The current counter value is 3
The current counter value is 2
The current counter value is 5
The final counter value was 6
```

6.3 An Alternative Build Procedure

In this example so far we built the two projects in the work space separately. However it is also possible to build them at the same time. To do this we have to clean up the current products of the projects and we can do this with the `clean.all` `ant` target.

```
[wile@amce]$ cd ..
[wile@amce]$ ant clean.all
```

You can then use the `lib.all` target to build both projects.

```
[wile@amce]$ ant lib.all
```

You should notice that the `icebucket` is built first as `gromit` depends on it. In fact most of the targets you can execute on a project have a "`.all`" version at the work space level which executes over all projects in the work space. The order in which the projects are executed is from least dependent to most dependent.

6.4 Logging Output

Given that the output of this program is the same as `Counter1` in `wallace` what is the difference? The main difference between the two programs is that Counter1 uses the `System.out` object to handle its output, while `Counter2` uses the `logging` pack-

age provided by the Apache Commons project (http://jakarta.apache.org/commons/index.html). This package is used in preference to `System.out` as it is much more flexible and configured. The first example of this configurability will be demonstrated in the following section. But before we get there lets take a quick look at how to use the logging package.

Most code written to use the logging package simply needs to obtain an instance of the Log interface and then use its methods to handle output String objects. The following line from the `Counter2` code shows how code can obtain an instance of the Log interface.

```
Log logger = LogFactory.getLog("icecube.tools.examples");
```

In this line the argument to the `getLog` method is the name of the package in which the class that shall be using it belongs. This is the recommended approach to obtaining instances of the Log interface, as it makes it easier to re-direct messages from different packages to different destinations. This issue will not be covered any further here.

Once an instance of the `Log` interface has been obtained it has six different levels of logging. Going from the least serious to the most serious these levels are the following.

- trace
- debug
- log
- warn
- error
- fatal

In `Counter2` you can see examples of using both the `log` level and the `fatal` level, some of which are shown in the following extracts.

```
logger.info("The current counter value is " +
        currentCounter);
```

```
systemlogger.fatal("Args should be: count [ interval [ reset ] ]");
```

The one other item to bear in mind is that every application should set up a consumer of the output `String` objects. In most cases, as in the `main` routine in `Counter2`, this is done by a call to the `installDefault` method of the `LoggingConsumer` class. This effectively sends any output to `System.out`. Any more detailed discussion beyond that covered in the next section is beyond the scope of this document.

The above discussion should be sufficient for basic logging within the IceCube software.

6.5 Testing Code

As noted in Section 3.3 ‘‘Testing Code’’, testing is an important part of a good software development cycle. The IceCube Software Development environment supports unit testing by making use of the JUnit (http://www.junit.org/index.htm) package.

In principle every class, except the most trivial ones, should have a matching test class. The development environment has been set up so that the test class is contained in the

`test` sub-package of the package containing the original class. For example, the following files in the `gromit` project contain the `Counter2` class and its test class.

```
src/icecube/tools/examples/Counter2.java
src/icecube/tools/examples/test/Counter2Test.java
```

The test class is designed to implement code that confirms that the public interface of the original class correctly fulfills its requirements. For example, the requirements for the Counter2 class are the following:

- It shall increment its count by one whenever requested.
- It shall be possible to reset the counter to zero.
- It shall log its status after its count has increase by a specified number.
- It shall not log any status if the specified interval is zero.
- Any change to the logging interval will take effect immediately, and if the new value is less than the interval since the last logging, then the interval will expire the next time the count increments.

Thus the Counter2Test class has the following five methods which tests each of these requirements.

- `testCounting`.
- `testReset`.
- `testInterval`.
- `testZeroInterval`.
- `testModifyInterval`.

To see these tests in action you can execute the `test ant` target in the project directory.

```
[wile@amce]$ ant test
```

This should report that five tests ran and there were zero failures or errors. The situation where there are errors is covered in Section 8.0 ‘‘Example: Delivering a Project’’. For now, if you want detailed information on the test that ran you can create some HTML output with the `report` ant target.

```
[wile@amce]$ ant report
```

The report target is the default target for a project, so you can achieve the same result by simply invoking `ant` without a target.

```
[wile@amce]$ ant
```

In either case to view the generated pages you need to point your favorite browser to the following file (with respect to the work space.)

```
docs/junit/index.html
```

By clicking on the various hyperlinks in this page you can explore the information that is available from the tests.

Once you have explored the information available from the test you are almost ready to move on to the next example except for cleaning up the work area. The `dispose` task of `bfd` is all you need to do this.

6.6 Review

In this example we built on the first example and introduced some more `ant` targets:

test: executes the unit tests for a project.

report: transforms the test results into HTML pages.

clean.all: executes the `clean` target for all projects in a work space.

lib.all: executes the `lib` target for all projects in a work space.

Along with this, the idea of using the logging system as a replacement for the `System.out` and `System.err` streams was introduced. This example also reviewed the infrastructure of the development environment that supports unit testing (This will be dealt with in detail in Section 9.0 ‘‘Example: Creating a Test’’).

7.0 Sidebar: Creating and Filling a Personal Archive

The examples from here on require you to be able to commit the modification you make back into the code archive. Clearly committing back to the main IceCube archive does not make sense as this would destroy the example files that are already in there. In order to be able to commit back to an archive you need to create your own personal archive and work from that one.

Since creating an archive is not really part of the development environment, I will simply go through the steps you need to execute without much detail. If you already have your own archive you can skip to the last part of this section which shows how to copy across the example project we will be using.

At present CVS is the choice of archive for IceCube and so that is the type of archive you will need to create. (It should be noted that you need version 1.11 of CVS or later to be able to use multiple archives transparently!) Creating a CVS archive is very easy. All `sh` users need to do to create an archive is the following:

```
[wile@amce]$ mkdir ~/cvsroot
[wile@amce]$ CVSROOT=~/cvsroot
[wile@amce]$ export CVSROOT
[wile@amce]$ cvs init
```

While `csh` users can use the same sequence of commands, replacing the CVSROOT assignment line and the `export` line with the following.

```
[gromit@wallaby] setenv CVSROOT ~/cvsroot
```

Having created the archive, you need to copy the example project into the archive. This is done with the following commands:

```
[wile@amce]$ mkdir tmp
[wile@amce]$ cd tmp
```

```
[wile@amce]$ cvs -d :ext:glacier.lbl.gov:/home/icecube/cvsroot \
                                          export -D tomorrow feathers
[wile@amce]$ cd feathers
[wile@amce]$ cvs import -m "Example Code" feathers ice3 V00-00-01
[wile@amce]$ cd ../../
[wile@amce]$ rm -fr tmp
```

You should now be ready to use your own archive.

8.0 Example: Delivering a Project

In this example we will start with a nearly completed project, complete it and then deliver it. We will also cover how to work using two different code archives.

8.1 Setting up the Work Space

The code on which you will be working in this example should be residing in your own code archive. If not, refer to Section 7.0 ‘‘Sidebar: Creating and Filling a Personal Archive’’ to see how to set up this state of affairs. Therefore, before you start, you should make sure that `CVSROOT` is set to the appropriate value. The following command, or something similar should take care of that:

```
[wile@amce]$ CVSROOT=~/cvsroot
[wile@amce]$ export CVSROOT
```

(From now on only the `sh` version of commands will be given, `csh` users should know, by now, how to transform them into `csh` commands.)

You should now be able to initialize your work space (if not, look back at the previous examples.) The project with which you are going to work, `feathers`, depends on `icebucket`, however, if you now try to check `icebucket` out using `bfd` you will get an error something like the following:

```
cvs checkout: cannot find module `icebucket' - ignored
```

This is caused by the fact that `icebucket` is in the IceCube archive and not your personal one. Therefore you need to use a modified `bfd` command to check out `icebucket`, namely you’ll have to use the `-a` option, as demonstrated in the following command:

```
[wile@amce]$ bfd co -a :ext:glacier.lbl.gov:/home/icecube/cvsroot \
                                                        icebucket
```

The `feathers` project also needs to be checked out into the work space, this time from your own archive, so the `-a` option is not required.

8.2 Tracking Down Problems

In principle the feathers project is exactly the same as the `gromit` project in the previous example. However, if you build and run it (noting that the class containing the main routine is now called `Counter3`) you should get the following output:

```
The current counter value is 1
The current counter value is 4
```

```
The current counter value is 3
The current counter value is 6
The final counter value was 6
```

This clearly does not match the output seen from `Counter2` in the gromit project. If we now run the unit tests for this project we can see that there are two failures, namely in the `testInterval` and `testModifyInterval` tests.

If you bring up the HTML bases results page, it will give you a complete trace back to where the failure was detected. In this case the `LoggingConsumer` (in the form of a `VerifiableAppender`) discovered that the output was not what was expected.

One point of interest at this time is the difference between a test failure and an error. A failure means that some expectation that was encoded in the test was not met (see Section 9.0 ‘‘Example: Creating a Test’’ for more details on how this is coded) while an error means that the test failed at a mechanical level, meaning the code in the test itself is not correct.

It is left as an exercise for the reader to find the deliberate error (any accidental ones you find do not count!) and fix it.

8.3 Archiving Your Changes

Eventually you will have found the problem with `feathers` and fixed it. Having done that then all the tests should execute successfully. If they do not, fix the code until they do. At this point in time you are ready to archive your changes. The `bfd archive` task is used to do this. This task has an optional argument, `-m`, that allows you to enter a brief comment about why this code is being checked back into the archive. The following command is an example of this approach.

```
[wile@amce]$ bfd archive -m"Fixed bug in setInterval method" feathers
```

Note that if you do not specify the option, then an editor will appear in which you can enter a longer description of the changes. By default `vi` is the editor. To change this define the environmental variable `EDITOR` to be whichever editor you prefer, e.g. `emacs`.

8.4 Delivering Your Changes

Now that you have a working version of feathers it is ready to be delivered to the next stage of development. For projects that are in the IceCube archive this could either be sub-system integration testing or entry into the continuous build system. Any delivered project is expected to pass all of its own tests and not cause any of the larger scale tests to fail. If a delivery does not satisfy these requirements then it must be backed-out.

Delivery of a project is done by the `deliver` option of the `bfd` system. When delivering a change it is necessary to either specify a tag that will uniquely identify the change or specify the type of change in which case a new “production” tag will be generated automatically (“production” tags take the form VXX-YY-ZZ). This means that one of the following options must be used to deliver a package.

-t <tag>: This option is used to provide a private tag or a production tag that can not be generated by one of the following options.

-b: This option is for the delivery of a "*b*ug" modification. Delivery of a "bug" modification normally increments the ZZ portion of a production tag1 and implies there have been no changes to the public API of the project.

-n: This option is for the delivery of a "mi*n*or" modification. Delivery of a "minor" modification increments the YY portion of a production tag and implies that there have been some changes to the public API of the project, but the main feature set remains unchanged

-j: This option is for the delivery of a "ma*j*or" modification. Delivery of a "major" modification increments the XX portion of a production tag and implies that there have been significant changes to the public API of the project, and that the main feature set has been changed.

In this example the change you should have made does not affect the public interface so this means we can consider this the delivery of a bug fix. The following demonstrates this delivery.

```
[wile@amce]$ bfd deliver -b feathers
```

The above command will require you to confirm that you want to deliver the project with the next appropriate production tag. (You should confirm this tag at this time.)

8.5 Review

This example demonstrated some more options of the `bfd` system. It dealt with how to check out projects which are stored in different archives, along with introducing the following other `bfd` options:

archive: enters the current version of a project (or file) into the code archive.

deliver: tags the current code as a stable tested version for use by others.

Also it was shown what happens when code does not pass a unit test.

9.0 Example: Creating a Test

This example builds on the previous example and introduces a new requirement on the Counter class. This requirement is captured as a new test and then the code is modified so that the test can successfully execute.

9.1 Setting up the Work Space

This example will build upon the work of the previous one, so you should already have a work space in which to work on this example. If not you'll need to set up a work space that contains the `icebucket` project and a working version of the `feathers` project taken from your own archive.

All the examples in the following sections will build upon the example that precedes them so, unless they state otherwise, you can simply use your existing work space from now on.

9.2 Adding new Requirements

After working through the previous example you may have realized that there is no requirement specifying what should happen when the logging interval is specified to be a negative number. The two requirements dealing with this behavior are the following:

- It shall log its status after its count has increased by a specified number.
- It shall not log any status if the specified interval is zero.

Clearly it does not make sense to have the interval specified by a negative number therefore we can add a new requirement to this class, namely the following one:

- It shall be an error if the interval is specified by a negative number.

A new requirement means that we need to write a new test. In this case it is easy. We simply want to try and set the interval to be a negative number and check that an `IllegalArgumentException` is throw by the `setInterval` method. Figure 1 shows how this test can be written.

FIGURE 1: The new test for the `Counter3` class.

```
/**
 * Test that a negative value for the interval is correctly rejected.
 */
public void testNegativeInterval
{
    try {
        testObject.setInterval(-1);
        fail("A negative value for the interval was not rejected.");
    } catch (IllegalArgumentException e) {
    }
}
```

The `fail` method is part of the test suite and is used to signal when a test should not have reached a certain point in the code. In this case the exception should have been thrown before this line of code so if the class is working correctly this line of code will not get executed.

Add the above code to the `Counter3Test.java` file and run the test target in the `feathers` project. You should see output similar to the following:

```
[junit] Tests run: 6, Failures: 1, Errors: 0, Time elapsed: 1.298 sec
```

This shows that the new test is running (as there are now six tests) and a check of the HTML output will show that it is the `testNegativeInterval` test that has failed.

9.3 Documenting the Code

In this case modifying the code to pass the test should be trivial and is therefore left as an exercise for the reader (don't you just hate that!). However, the job of satisfying a new requirement is not complete until this new behavior is documented. In this case, while an `IllegalArgumentException` is a runtime exception, and so is not a checked exception, this modification has changed the behavior of the `setInterval`

method and so should be documented in its Javadocs (For those reader not familiar with this from of self-documenting code check out the Javadocs home page - http://java.sun.com/j2se/javadoc/index.html). To do this the following line should be added to that method's javadocs just below the `@return` tag.

```
* @throws IllegalArgumentException if the interval is a negative number.
```

The new javadocs can be generated using the `javadocs` target for `ant`. The following command demonstrated this:

```
[wile@amce]$ ant javadocs
```

To view the generated pages you need to point your favorite browser to the following file (with respect to the work space.)

```
docs/feathers/api/index.html
```

You will notice that these documents only include the classes and package in the `feathers` project. To include all the projects in a work space, i.e. to include `icebucket` in this case, you need to execute the javadocs.ws target in the work space directory, as shown in the following command:

```
[wile@amce]$ ant javadocs.ws
```

The generated pages can be viewed from the following page.

```
docs/api/index.html
```

Whichever way you generate your javadocs you should now see that the new line is in the `setInterval` description.

9.4 Delivering Changes to the Public API

In principle this change has changed the interface to the `Counter3` class so once the code has passed all of its tests, old and new, it should be delivered as a minor release. That can be done with the following commands:

```
[wile@amce]$ bfd ar -m"Implemented negative interval requirement" \
                                                              feathers
[wile@amce]$ bfd deliver -n feathers
```

9.5 Review

In this example we have seen how to include new requirements by creating unit tests to check that the requirement is met. We have also seen how to produce code documentation for both a single project and an entire work space using javadocs.

10.0 Example: Starting a new Project

In the next few examples we will create a new package, `mcgraw`, that will contains a simple tools with which we can exercise the `Counter3` class in `feathers`.

In this example we shall see how you can start this new project.

10.1 Creating the Project's Archive Entry

It is far easier to have a new project started in the archive first, rather than develop the project and then get it added to the archive, though that is possible. The only way to get a new project entry added to the IceCube code archive is to request the Librarian make an entry for you. However, in this example we will continue to use your personal archive and so you can create your own entry. Again, this is not really part of the development environments responsibility so it will not be discussed any further than to say you need to execute the following command (assuming `CVSROOT` points to your archive):

```
[wile@amce]$ mkdir $CVSROOT/mcgraw
```

10.2 Populating a new Project

Once the new project's entry has been made in the archive you can check it out just like any other project. However, in this case you will end up with an empty directory (with the exception of any archive management files, such as the `CVS` directory!). Before you can populate this empty directory you need to decide which package will be the default package for the project. The default package is the one in which any newly created class or interface will be placed unless it is explicitly stated otherwise. The details of this will be dealt with in Section 11.0 ''Example: Adding a new Interface''. In this example `icecube.tools.examples.mcgraw` will be the default package.

To populate the new project you need to start in the work space directory and execute the following command:

```
[wile@amce]$ ant -DPACKAGE=icecube.tools.examples.mcgraw \
                                          -DPROJECT=mcgraw createProject
```

As you should be able to see, this command has created a number of files and directories. Figure 2 shows their structure.

FIGURE 2: The contents of the new `mcgraw` project.

```
mcgraw
    +--- build.xml
    +--- project.xml
    +--- src
    |        +--- icecube
    |                +--- tools
    |                        +--- examples
    |                                +--- mcgraw
    |                                        +--- package.html
    |                                        +--- test
    |                                                +---- package.html
    + resources
         +---- test
```

- The `build.xml` file is used by `ant` to build this project and is a copy of a standard file. You should normally have no reason to change this file.
- The `project.xml` file describes the project and is used by the standard `build.xml` file. For the moment the default file will be sufficient, but this is a file

that will need editing when we add to the project in Section 12.0 ‘‘Example: Creating a new Package’’.

- The `src` directory holds the java source and documentation files. The subdirectories below `src` are set out according to the packages which are contained in the project. As you can see in Figure 2 each package has a test sub-package created in which its test source files are kept. Also, you will notice that each package contains a `package.html` file that contains an HTML description of the package that is used to create the javadocs for that package. The default `package.html` in the `test` package does not need changing, but the file in `mcgraw` does. The following few lines can be used to replace the text between the `<body>` tags.

```
<p>
Provides classes and interfaces that implement a simple predicate
mechanism.
</p>
```

- Any file in the `resources` directory that is not under the `test` sub-directory will be copied into this project jar file when it is created, while files under the `test` sub-directory will be copied in this project’s test jar file.

10.3 Archiving the new Project

Now that we have created these files we need to make sure they get archived as part of the project. To make this happen the archive needs to be told that it should include these files. This is done by the `add` or `uadd` option of `bfd`. The `add` option allows you to specify a single file to be added, while the `uadd` option adds all unknown files in a project. In this case we’ll use the `uadd` option. This option recursively adds directories so you will be prompted to confirm each new set of directories and files. Always make sure before confirming each add that there are no extra work or temporary files in the list of files to be added. The following command adds all the new files in `mcgraw` into the archive.

```
[wile@amce]$ bfd uadd mcgraw
```

Now that these files have been added they can be archived and the project is ready for use.

10.4 Review

In this example we have seen how to populate a new, empty project. We have seen the basics files and directory structure that any java project should have, and we have found out how to add files to the set of files being archived for a project.

11.0 Example: Adding a new Interface

In this example we will create an interface and along with that interface we will create its unit test and demonstrate some of the basic techniques that should be used when coding up a test for the interfaces.

11.1 Creating the Skeletons for a new Interface

The interface we are going to create is a simple predicate that takes as its argument an int and returns a boolean based on the logic programmed into the implementation of the interface. This translates into the following requirement:

- The interface shall return `true` if the integer satisfies its predicate, otherwise it will return false.

As we now have the requirement for the interface we are ready to create it. This is done with the `createInterface` class in `ant`. The following command shows how this can be done (make sure that either `mcgraw` is set up as the default project or that you execute this command in the `mcgraw` project directory):

```
[wile@amce]$ ant -DCLASS=IntPredicate createInterface
```

You should see that a file has been created in both the default package and its test subpackage, namely `IntPredicate.java` and `IntPredicateNoRunTest.java`. The name of the test class end with `NoRunTest` as an interface needs a concrete implementation to be able to execute its tests. Therefore this class is not run by the development, but is available for classes which implement the interface to use in their tests. An example of this will be shown in Section 13.0 ‘‘Example: Adding a new Class’’.

11.2 Creating the Interface’s Tests

As noted in the previous section, the test for an interface is not the same as that for a concrete class. This means that there are few features which appear in this type of test that will not necessarily appear in the test of a concrete class. If we look at the source code for `IntPredicateNoRunTest.java` we immediately see one of these features, namely the `setIntPredicate` method. This is used by any test of a class implementing this interface to set up the test object for the interface tests. As you can see this is already written for you in the skeleton that the `createInterface` target produced, therefore we do not need to worry about this until Section 13.0 ‘‘Example: Adding a new Class’’.

The other features of interface tests are functions that allow these other tests to set up conditions before the interface test is run. The easiest way to explain this is to continue with the example and look at the code that makes up the tests we need for this interface. Figure 3 shows the two tests which you should add to the `IntPredicateNoRunTest.java` file, replacing the `testSomthing` method.

As you can see from these tests, each calls a `prepare` method before executing the meat of the test. These are abstract methods that any subclass of the test should implement to set up the right conditions before the test. In the case of this particular test the setup is simply setting an appropriate value for `value`. To be able to do this a protected `setValue` method is needed to allow subclasses to set this variable. Figure 4 shows this method that, along with the two declarations of the necessary `prepare` methods, needs to be added to the `IntPredicateNoRunTest.java` file just after the `setIntPredicate` method. You will also need to add the `value` instance variable. Figure 5 shows this declaration which should be placed just after the `testObject` declaration. The one other modification you will need to make to this file is to declare the `IntPredicateNoRunTest` class to be abstract.

FIGURE 3: The two tests to be added to `IntPredicateNoRunTest.java`

```
/**
 * Test that the predicate returns true if the integer satisfies the
 * predicates logic.
 */
public void testAffirmed()
{
    prepareAffirmed();
    assertTrue(testObject.isAffirmed(value));
}

/**
 * Test that the predicate returns false if the integer fails to satisfy
 * the predicates logic.
 */
public void testDenied()
{
    prepareDenied();
    assertFalse(testObject.isAffirmed(value));
}
```

FIGURE 4: The `set` and `prepare` methods that need to be added to `IntPredicateNoRunTest.java`

```
/**
 * Sets the value that will be used in the test.
 */
protected void setValue(int value)
{
    this.value = value;
}

/**
 * Prepare the test for an affirmative result to the predicate.
 */
protected abstract void prepareAffirmed

/**
 * Prepare the test for an denied result to the predicate.
 */
protected abstract void prepareDenied();
```

FIGURE 5: The `value` declaration that needs to be added to `IntPredicateNoRunTest.java`

```
/** The value that will be used in the test. */
private int value;
```

Hopefully the tests themselves should be fairly transparent. The `assertTrue` function is part of the testing framework and if its argument does not evaluate to `true` then the test will fail. The testing framework has a number of different `asset` style methods and

you should chose the one appropriate to the type of comparisons you need on a test by test basis.

11.3 Creating the interface

The nice thing about establishing the requirements for an interface (or class for that matter) first, and encoding them in a unit test, is that now the interface has pretty much written itself. For the tests to compile we must add the `isAffirmed` method declaration to the `IntPrecicate.java` file to get the tests to compile and the job is done. (In the case of classes we need to implement the methods needed by the test, but otherwise the process is much the same.)

Figure 6 shows the code to add, as an instance member method, to the `IntPrecicate.java` file.

FIGURE 6: The method declaration that needs to be added to `IntPredicate.java`

```
/**
 * Returns true if the integer satisfies its predicate, otherwise returns
 * false.
 *
 * @return true if the integer satisfies its predicate, otherwise returns
 * false.
 */
boolean isAffirmed(int value);
```

Having added this declaration, the `mcgraw` project should now compile and be ready for archiving. Don't forget to add the two new files to the project before archiving it, either with the `add` or `uadd` options for `bfd`.

11.4 Review

In this example we have worked through the process of developing a new interface by defining its requirements and capturing these in a set of unit tests. Then, by creating an interface which allows the tests to compile we have, by construction, the effective declaration of the interface.

12.0 Example: Creating a new Package

The next step after creating an interface is often to create at least one implementation of that interface. In this example we will add a new package to the `mcgraw` project, in preparation for creating such an implementation in Section 13.0 "Example: Adding a new Class".

12.1 Adding another Package to a project

The package which will contain at least the first implementation of the `IntPredicate` interface will be the `impl` sub-package of the `mcgraw` package. To create the package you can execute the following command in the project directory:

```
[wile@amce]$ ant -DPACKAGE=icecube.tools.examples.mcgraw.impl \
                                                        createPackage
```

The results of this command are similar to the results of creating a new project as that command crates a default package. The new package contains a test sub-package and `package.html` files for both the new package and its `test` sub-package. As before the `package.html` file should be edited to contain a description, however brief, of the package and its purpose. Figure 7 shows a suggestion for the contents of the `<body>` element in the new package's `package.html` file.

FIGURE 7: The suggested contents of the `<body>` element of the `impl` package's `package.html` file

```
<p>
Provides implementations of interfaces contained in the
{@link icecube.tools.examples.mcgraw} package.
</p>
```

The final part of adding a package to a project is to add it to the project.xml file as part of the `<projects>` element. In this particular case you simply need to add the following line to the `<projects>` element:

```
icecube.tools.example.mcgraw.impl
```

The new files created by this process should now be added to the project and archived.

12.2 Review

The example has demonstrated the process of adding a new package to an existing project. The default files can be created by running `ant`, and then the package needs to be added to the project's `project.xml` file.

13.0 Example: Adding a new Class

In this example we will create a simple implementation of the `IntPredicate` interface that we created in Section 11.0 ''Example: Adding a new Interface''.

13.1 Creating the Skeletons for a new Class

Creation of the skeleton files for a class is almost identical to creating these types of files for an interface, which we did in Section 11.0 ''Example: Adding a new Interface'', with the exception that you use the `createClass ant` target rather than the `createInterface` one. Also, in this particular case, as the new class is not targeted for the default package of the projects, we need to state the package in which it is to be created. How this is all done is shown in the following command, executed in the project's directory:

```
wile@amce]$ ant -DPACKAGE=icecube.tools.examples.mcgraw.impl \
                                        -DCLASS=MultipleOfThree createClass
```

This should have created both of the source files, `MultipleOfThree.java` and `MultipleOfThreeTest.java`, in the appropriate directories.

13.2 Creating the Classes's Tests

As `MulitpleOfThree` is going to be an implementation of the `IntPredicate` interface its tests should include all the tests for `IntPredicate`. The current method for doing this is to subclass the `IntPredicateNoRunTest` class. (Currently multiple inheritance it handled by creating a test class for each interface, a better approach is currently under development.) Therefore, the first step is to modify the inheritance of the `MultipleOfThreeTest` class so that it now inherits from `IntPredicateNoRunTest`, rather than `TestCase`.

We now have to make `MultipleOfThreeTest` a concrete class by defining implementations of the abstract `prepare` methods of `IntPredicateNoRunTest`. Figure 8 shows the implementation of these methods. It should be noted that these methods use constants as their values. This will be explained later in Section 13.3 ''The Use of Constants''.

FIGURE 8: The implementations of the `prepare` methods in `MultipleOfThreeTest`.

```
protected void prepareAffirmed()
{
    setValue(PASS_VALUE);
}

protected void prepareDenied()
{
    setValue(FAIL_VALUE);
}
```

For the `IntPredicateNoRunTest` methods to be able to execute we need to set their `testObject` to be an instance of the `MulitpleOfThree` class. This can be done in the `setUp` method which is executed before each test. The matching `tearDown` method is executed after each test and provides an opportunity to clean up after every test. Figure 9 shows how these twos method can be implemented for this example.

As well as using the tests that apply to `IntPredicate`, it is often necessary, or simply useful, to add extra tests to check that a class is behaving correctly. In this example we will add one more test, `testMultiplesOfThree`, just to explore a little bit of the possible phase space of integers and check that the `MulitpleOfThree` class behaves as expected. Figure 10 shows the implementation of this test. (This should replace the `testSomthing` method in `MultipleOfThreeTest.java`.)

13.3 The Use of Constants

As has already been noted, the tests in this section have been written in terms of constants. The reason for this is that this approach improves the readability of the tests (and other code when you write it) by avoiding the appearance of "magic numbers" in the middle of the code, and offering the opportunity to describe the nature of the number being used. Figure 11 shows the declarations of the constants that need to be added to `MultipleOfThreeTest`, along with their descriptions.

FIGURE 9: The `setUp` and `tearDown` methods for `MultipleOfThreeTest`.

```
/**
 * Sets up the fixture, for example, open a network connection.
 * This method is called before a test is executed.
 */
protected void setUp()
{
    testObject = new MultipleOfThree();
    setIntPredicate(testObject);
}

/**
 * Tears down the fixture, for example, close a network connection.
 * This method is called after a test is executed.
 */
protected void tearDown()
{
    testObject = null;
    setIntPredicate(testObject);
}
```

FIGURE 10: The implementation of the `testMultiplesOfThree` in `MultipleOfThreeTest`.

```
/**
 * Test some of the possible phase space of integers to check that the
 * predicate behaves as expected.
 */
public void testMultipleOfThree()
{
    final int finished = TEST_VALUES.length;
    for (int value = 0;
         finished != value;
         value++) {
        assertEquals(TEST_RESULTS[value],
                testObject.isAffirmed(TEST_VALUES[value]));
    }
}
```

13.4 Creating the Class

With the test now completed we can turn our attention to the `MulitpleOfThree` class. The skeleton of this class needs very little to be changed to make it work. First, the class is an implementation of `IntPredicate` so this needs to be added to the declaration of the class. Next, as this class can be built without any parameters the default constructor should be made public rather than private which is the access level at which the skeleton is created. (You should also drop the last two lines of the constructors javadocs when you do this.) Finally the class needs an implementation of its `isAffirmed` method. Figure 12 shows a suitable implementation of this method.

FIGURE 11: The declaration and definition of the constants used in `MultipleOfThreeTest`.

```
/** A value which will be affirmed by the predicate. */
private static final int PASS_VALUE = 3;

/** A value which will be denined by the predicate. */
private static final int FAIL_VALUE = 2;

/** A set of values to test against the predicate. */
private static final int[] TEST_VALUES = new int[]{0, 1, 4, 7, 9, 15, 16,
                                                   20, 21
/** A set of results for the values in TEST_VALUES. */
private static final boolean[] TEST_RESULTS = new boolean[]{true, false,
                            false, false, true, true, false, false, true};
```

FIGURE 12: The implementation of the `isAffirmed` in `MultipleOfThree`.

```
public boolean isAffirmed(int value)
{
    return 0 == (value % 3);
}
```

You should now be able to successfully run all the tests for the `MulitpleOfThree` class in which case you can add and archive this code.

The completion of the `MulitpleOfThree` class also completes our development of the `mcgraw` project. (Of course you are welcome to add other implementations of the `IntPredicate` interface to this project.) This means that we should deliver this project as a completed product. This is done with the following command (executed in the workspace directory):

```
wile@amce]$ bfd deliver -j mcgraw
```

13.5 Review

In this example we saw how to create a new class and implement its tests. We also saw how to deliver a completed project by incrementing the major field of its tag.

14.0 Example: Added A Project Dependency

In this example we will modify the original `Counter3` class in the `feathers` project to use the `IntPredicate` interface and count the number of integers which are "affirmed" by the predicate. This change will make the `feathers` project have a direct dependency on the `mcgraw` project. We will capture this dependency in the `project.xml` file so that `feathers` can be built correctly.

14.1 Modifying `Counter3`

We are going to modify `Counter3` so that its `main` routine takes, as it first argument, the name of a class that implements the `IntPredicate` interface. An instance of this

class will then be used in the loop portion of the method to decide whether or not to increment the counter.

To begin with we need to change the expected argument list. To do this `MIN_ARG` needs to be changed to be 2, as we need a class name and a count to run this method. We then need to add "predicate_class" to the usage string before the "count" argument.

Next we need to be able to instantiate an instance of the `IntPredicate` interface. Figure 13 shows the code that will do this. This code should be added after the declaration and assignment of the `arg` variable.

FIGURE 13: The code to instantiate an instance of the `IntPredicate` interface.

```
IntPredicate predicate = null;
try {
    predicate = (IntPredicate) Class.forName(args[arg]).newInstance();
catch (ClassNotFoundException e1) {
    systemlogger.fatal(e1.toString());
    return;
catch (InstantiationException e2) {
    systemlogger.fatal(e2.toString());
    return;
} catch (IllegalAccessException e3) {
    systemlogger.fatal(e3.toString());
    return;
}
arg++;
```

All that is then needed is to place an `if` statement around the `counter.count()` statement so that the counter only increments when the predicate is "affirmed". Once we have done this we are ready to try and compile the new version of `Counter3`.

14.2 Adding a Dependency to a Project

If you try to compile the new version of `Counter3` you will quickly discover that it complains that the `IntPredicate` interface does not exist. The reason for this is that we have not yet told the build system that `feathers` is now dependent upon `mcgraw`. To do this you need to edit the `project.xml` file in `feathers` and include `mcgraw.jar` in the same element that contains `icebucket.jar` (separating them by whitespace). Now you should be able to build `feathers` without a problem.

Just to make certain that everything builds correctly, the following commands, executed in the work space directory, clean up all the files generated so far and then builds and tests all the projects in the work space, in the correct order.

```
[wile@amce]$ ant clean.ws
[wile@amce]$ ant
```

If you now try run the new method using the following command:

```
[wile@amce]$ java -cp lib/feathers.jar \
             icecube.tools.examples.Counter3 \
             icecube.tools.examples.mcgraw.impl.MultipleOfThree 100 10
```

you should get the following output:

```
The current counter value is 10
The current counter value is 20
The current counter value is 30
The final counter value was 34
```

You should notice that you only need to specify the `feathers.jar` file in the classpath as `mcgraw.jar` and `icebucket.jar` are automatically added to the classpath because of the declared dependencies.

At this point in time you have now finished with your changes to feathers and you can archive the new version of `Counter3` and deliver this as a major version using the following command:

```
wile@amce]$ bfd deliver -j feathers
```

14.3 Review

In this example we have seen how a dependency of one project on another is captured in the `project.xml` file and that this can get picked up automatically and put into the JVMs classpath.

15.0 Review of The Standalone Personal Software Development Environment

In the preceding sections we have seen how the personal software development environment can help with many aspects of software development from the management of a developers work space to the delivery of a completed software project. By working through these examples you should be familiar and comfortable with using this environment to develop IceCube software.

Part II: The baselined Personal Software Development Environment

16.0 To be completed in Version 2

Part III: The “Collaborative” Software Development Environment

17.0 To be completed in Version 2

Appendix A: Installing the bfd System by Hand

The bfd system should be installed when the IceCube software distribution is installed on a machine along with the necessary environment settings. However, if this is not available then this appendix outlines how this system can be installed by hand.

A.1 Before installing bfd

Before we can start you need some sort of unix installation. This environment has been tried under Linux, MacOS X and cygwin, but is general enough that it should work on other flavors too, but there is no guarantee. Apart from the standard unix tools such as cvs, zip, etc., you will need to have both python 2.2+ and Java 1.4+ installed. If you do not then check out the following URLs:

http://www.python.org/http://java.sun.com/j2se/

A.2 Downloading bfd

To install a local copy of bfd, start by moving to the directory in which you want it placed. The following is an example of where you may want to place the files:

```
[wile@amce]$ cd $HOME/bin
```

Next you need to make sure that the right environmental variables are set up to access the IceCube archive. For sh users the following commands will do this

```
[wile@amce]$ CVSROOT=:ext:glacier.lbl.gov:/home/icecube/cvsroot
[wile@amce]$ export CVSROOT
[wile@amce]$ CVS_RSH=ssh
[wile@amce]$ export CVS_RSH
```

For csh users the following are necessary:

```
[gromit@wallaby] setenv CVSROOT \
                              :ext:glacier.lbl.gov:/home/icecube/cvsroot
[gromit@wallaby] setenv CVS_RSH ssh
```

After these are set you need to “export” the bfd project from the archive into the current directory with the following command:

```
[wile@amce]$ cvs export -D tomorrow bfd-tools
```

(The tomorrow option is used to guarantee that you get the latest version of the code.)

If you have an ssh key set up on glacier.lbl.gov then you should not need to do anything extra to get all the bfd files. However, if you do not have such a key installed you will be prompted for your password, which you should provide.

A.3 Setting up the Local Environment

Now you need to set up the necessary links and variables to run the `bfd` program. This requires setting up a soft-link, placing this link into your `PATH` variable and specifying a "root" directory. For `sh` users these tasks can be done at the prompt using the following:

```
[wile@amce]$ ln -s bfd-tools/bfd.py bfd
[wile@amce]$ PATH=${PATH}:${HOME}/bin
[wile@amce]$ export PATH
```

Meanwhile `csh` users need to execute the following:

```
[gromit@wallaby] ln -s bfd-tools/bfd.py bfd
[gromit@wallaby] setenv PATH ${PATH}:${HOME}/bin
```

(Note that if `~/bin` is already in your `PATH` variable you do not need to execute the line which adds it to this variable). These environmental setting will be copied into the appropriate setup files that are created by the `bfd` system when it initializes work space.

Appendix B: Installing the IceCube tools Directory by Hand

Access to the third party tools that are used by the IceCube software is done via the `tools` directory in either the work space or a baseline. The IceCube software distribution contains all of the third party distributions needed by IceCube software. However, if this is not installed then this appendix outlines how the tools can be installed by hand. Currently, the default `bfd` scripts, and thus IceCube's, are set up to handle to following packages:

Ant: a dependency and build system.

JUnit: a unit testing framework for Java.

Xalan: an eXtensible Stylesheet Language Transform processor.

Commons-Logging: a generic interface to different logging solution for Java.

Log4j: a complete logging solution for Java.

Jython: an interactive scripting tool for Java.

Before downloading the tools used by IceCube a suitable directory needs to be created, something like `/home/icecube/tools` is recommended. There are two ways to fill this directory: the easy way; and the hard way.

B.1 The Easy Way

The latest version of all the tools can be found on `glacier.lbl.gov` and can be downloaded and installed with the following commands (where `<user>` is replaced with your user name on `glacier`).

```
[wile@amce]$ cd /home/icecube/tools
[wile@amce]$ scp \
    <user>@glacier.lbl.gov:/home/icecube/tools/distributions/latest.tgz .
[wile@amce]$ tar xvzf latest.tgz
```

(Note: On MacOS X you should use the `gnutar` command)

B.2 The Hard Way

If, for some reason, the distribution on glacier does not match your needs then you may have to download each individual tool from its home web site. This appendix covers the versions that were in use in January 2003. In most cases a simple change in the version numbers where they appear is sufficient to make these instructions work for the current versions. The following file, specified with respect to the directory in which the `bfd-tools` directory was created (See Appendix A: "Installing the `bfd` System by Hand"), contains the list of the current versions:

```
bfd-tools/bfdroot/scripts/tools_vers.sh
```

By convention a subdirectory named `packaged` should also be created in the tools directory, which will be the destination of the packaged tools files when they are down-

loaded. The tools should now be downloaded into the packaged directory. Table 1 to Table 6 give the download information each of the tools.

TABLE 1: Download information for ant 1.51

Web Site	http://jakarta.apache.org/ant/
Download URL	http://ant.apache.org/old-releases/v1.5.1/bin/jakarta-ant-1.5.1-bin.zip

TABLE 2: Download information for JUnit 3.8.1

Web Site	http://www.junit.org/
Download URL	http://download.sourceforge.net/junit/junit3.8.1.zip

TABLE 3: Download information for Xalan 2.4.1

Web Site	http://xml.apache.org/xalan-j/
Download URL	http://xml.apache.org/dist/xalan-j/xalan-j_2_4_1-bin.tar.gz

TABLE 4: Download information for Commons-Logging 1.0.3

Web Site	http://jakarta.apache.org/commons/logging.html
Download URL	http://www.apache.org/dist/jakarta/commons/logging/binaries/commons-logging-1.0.3.tar.gz

TABLE 5: Download information for Log4j 1.2.7

Web Site	http://jakarta.apache.org/log4j/
Download URL	http://jakarta.apache.org/log4j/jakarta-log4j-1.2.7.tar.gz

TABLE 6: Download information for Jython 2.1

Web Site	http://www.jython.org/
Download URL	http://prdownloads.sourceforge.net/jython/jython-21.class?use_mirror=unc

Once downloaded each tool should be installed. For those files with the `.tar.gz` suffix the following command, when executed in the `packaged` sub-directory, will unpack and install them:

```
[wile@amce]$ gunzip -cf <file> | tar -C .. -x
```

(Note: On MacOS X you should use the `gnutar` command.)

Files with the `.zip` suffix can be handled with the following command:

```
[wile@amce]$ unzip -d .. <file>
```

To install Jython you need to execute the class file, which can be done with the following command:

```
[wile@amce]$ java -cp . jython-21 -o Jython-2.1 demo lib source
```

(Note that in this case the `.class` suffix should *not* be specified.)

Now all the necessary files should be in the chosen tools directory and so you can proceed with the use of the `bfd` system.